White-box attack resistant cryptography

Hiding cryptographic keys against the powerful attacker

Dušan Klíneč, Petr Švenda {xklinec, svenda}@fi.muni.cz
Outline

• Short intro to symmetric/asymmetric cryptography
• Classical implementations & related problems
• CEF&CED, practical problems
• Whitebox cryptography, whitebox-AES
• Available implementations & attacks
• Future work, related R&D at CROCS@FIMU
Protecting key material for cryptographic functions

TROUBLES WITH KEYS
Cryptography with symmetric key

• Algorithms: DES, AES, Blowfish, IDEA, RC4...
• Key lengths: 16-32 bytes
• Usage: encryption, cryptographic checksum, auth.
Asymmetric cryptography - encryption

- Algorithms: RSA, Diffie-Hellman, ECC...
- Key lengths: 32 bytes (ECC-256) – 256 bytes (RSA-2048)
- Usage: encryption, digital signature, authentication
Standard vs. whitebox attacker model
Standard AES API (PolarSSL)

/**
 * 
 * AES key schedule (encryption)
 */

int aes_setkey_enc(aes_context *ctx,
                   const unsigned char *key,
                   unsigned int keysize);

/**
 * AES-ECB block encryption/decryption
 */

int aes_crypt_ecb(aes_context *ctx,
                   int mode,
                   const unsigned char input[16],
                   unsigned char output[16]);
Advanced Encryption Algorithm

Repeat 10 times
void simpleAES() {
    unsigned char key[32];
    unsigned char buf[16];
    aes_context ctx;

    memset(buf, 1, sizeof(buf));
    memset(&ctx, 0, sizeof(ctx));

    // Set the key
    sprintf((char*)key, "%s", "SecurePassword:nbu123");
    aes_setkey_enc(&ctx, key, 128);

    printf("Input: ");
    for (int i = 0; i < AES_BLOCK_SIZE; i++) printf("%2x", buf[i]);
    printf("\n");

    // Encrypt one block
    aes_crypt_ecb(&ctx, AES_ENCRYPT, buf, buf);
    printf("Output: ");
    for (int i = 0; i < AES_BLOCK_SIZE; i++) printf("%x", buf[i]);
}

OllyDbg – key value is static string
OllyDbg – key is visible in memory
What if AES usage is somehow hidden?
Whitebox attacker model

• The attacker is able to:
  – inspect and disassemble binary (static strings, code...)
  – observe/modify all executed instructions (OllyDbg...)
  – observe/modify used memory (OllyDbg, memory dump...)

• How to still protect value of cryptographic key?

• Who might be whitebox attacker?
  – Mathematician (for fun)
  – Security researcher / Malware analyst (for work)
  – DRM cracker (for fun&profit)
  – ...

Classical obfuscation and its limits

- Time-limited protection
- Obfuscation is mostly based on obscurity
  - add bogus jumps
  - reorder related memory blocks
  - transform code into equivalent one, but less readable
  - pack binary into randomized virtual machine
  - ...
- Barak’s (im)possibility result (2001)
  - family of functions that will always leak some information
  - but practical implementation may exists for others
Computation with Encrypted Data and Encrypted Function

CEF&CED
Scenario

• We’d like to compute function F over data D
  – secret algorithm F or sensitive data D (or both)
• Solution with trusted environment
  – my trusted PC, trusted server, trusted cloud…
• Problem: can be cloud or client really trusted?
  – server hack, DRM, malware…
• Attacker model
  – controls execution environment (debugging)
  – sees all instructions and data executed
CEF

- Computation with Encrypted Function (CEF)
  - A provides function F in form of P(F)
  - P can be executed on B’s machine with B’s data D as P(D)
  - B will not learn function F during computation
CED

- Computation with Encrypted Data (CED)
  - B provides encrypted data D as E(D) to A
  - A is able to compute its F as F(E(D)) to produce E(F(D))
  - A will not learn D
CED via homomorphism

1. Convert your function into circuit with additions (xor) and multiplications (and) only
2. Compute addition and/or multiplication “securely”
   – an attacker can compute \( E(D1+D2) = E(D1)+E(D2) \)
   – but will learn neither \( D1 \) nor \( D2 \)
3. Execute whole circuit over encrypted data
   • Partial homomorphic scheme
     – either addition or multiplication is possible, but not both
   • Fully homomorphic scheme
     – both addition and multiplication (unlimited)
Partial homomorphic schemes

• Example with RSA (*multiplication*)
  – \( E(d_1).E(d_2) = d_1^e \cdot d_2^e \mod m = (d_1d_2)^e \mod m = E(d_1d_2) \)

• Example Goldwasser-Micali (*addition*)
  – \( E(d_1).E(d_2) = x^{d_1r_1^2} \cdot X^{d_2r_2^2} = x^{d_1+d_2(r_1r_2)^2} = E(d_1 \oplus d_2) \)

• Limited to polynomial and rational functions

• Limited to only one type of operation (*mult* or *add*)
  – or one type and very limited number of other type

• Slow – based on modular mult or exponentiation
  – every operation equivalent to whole RSA operation
Fully homomorphic scheme (FHE)

- Holy grail - idea proposed in 1978 (Rivest et al.)
  - both addition and multiplication securely
- But no scheme until 2009 (Gentry)!
  - based on lattices over integers
  - noisy FHE usable only to few operations
  - combined with repair operation
Fully homomorphic scheme - usages

- Outsourced cloud computing and storage (FHE search)
  - Private Database Queries
  - using Somewhat Homomorphic Encryption
  - protection of the query content
- Secure voting protocols (yes/no + sum)
- Protection of proprietary info - MRI machines
  - very expensive algorithm analyzing MR data, HW protected
  - central processing restricted due to processing of private patient data
- Read more about current state of FHE
Fully homomorphic scheme - practicality

- Not very practical (yet 😊) (Gentry, 2009)
  - 2.7GB key & 2h computation for every repair operation
  - repair needed every ~10 multiplication

- FHE-AES implementation (Gentry, 2012)
  - standard PC ⇒ 37 minutes/block (but 256GB RAM)
Protection of cryptographic primitives

WHITEBOX RESISTANT CRYPTO
White-box attack resistant cryptography

• Problem limited from every cipher to symmetric cryptography cipher only
  – protects used cryptographic key (and data)
• Special implementation fully compatible with standard AES/DES… 2002 (Chow et al.)
  – series of lookups into pre-computed tables
• Implementation of AES which takes only data
  – key is already embedded inside
  – hard for an attacker to extract embedded key
Impractical solution

• Secure, but $2^{128} \times 16$B memory storage
WBACR AES – some techniques

- Pre-compute table for all possible inputs
  - practical for one 16bits or two 8bits arguments table with up to $2^{16}$ rows (~64KB)
  - \texttt{AddRoundKey}: data $\oplus$ key
    - 8bit argument data, key fixed
- Pack several operations together
  - \texttt{AddRoundKey+SubBytes}: $T[i] = S[i \oplus \text{key}]$;
- Protect intermediate values by random bijections
  - removed automatically by next lookup
  - $X = F^{-1}(F(X))$
  - $T[i] = S[F^{-1}(i) \oplus \text{key}]$;
AES – short remainder (used ops)

Whitebox cryptography - intro
Whitebox cryptography lifecycle

- [Secure environment]
  1. Generate required key (random, database...)
  2. Generate WAES tables (in secure environment)
- [Potential insecure environment]
  3. Compile WAES tables into target application
- [Insecure environment (User PC)]
  4. Run application and use WAES as usual (with fixed key)
Whitebox cryptography - intro

makeTable()

precompTable

AES

key

Environment outside control of an attacker

Environment under control of an attacker

data

encrypt(data)

encrypted data

key

Environment under control of an attacker
Resulting implementation

- More difficult to detect that crypto was used
  - no fixed constants in the code
  - precomputed tables change with every generation
  - even two tables for same key are different
  - (but can still be found)
- Resistant even when precomputed tables are found
  - when debugged, only table lookups are seen
  - key value is never manipulated in plaintext
  - transformation techniques should provide protection to key embedded inside tables
WBACR AES - pros

- Practically usable
  - implementation size ~800KB (tables)
  - speed ~MBs/sec (~6.5MB/s vs. 220MB/s)
- Hard to extract embedded key
  - Complexity semi-formally guaranteed
  - (if the scheme is secure)
- One can simulate asymmetric cryptography!
  - implementation contains only encryption part of AES
  - until attacker extracts key, decryption is not possible
WBACR AES - cons

• Implementation can be used as oracle (black box)
  – attacker can supply inputs and obtain outputs
  – even if she cannot extract the key
  – (can be partially solved by I/O encodings)
• Problem of secure input/output
  – protected is only AES, not code around
• Key is fixed and cannot be easily changed
• Successful cryptanalysis for several schemes
  – several former schemes broken
  – new techniques proposed
Can whitebox transform replace secure hardware (e.g., smart card)?

- Only to limited extent
- Limitation of arguments size
- Operation atomicity
  - one cannot execute only half of card’s operations
- No secure memory storage
  - no secure update of state (counter)
- Both can be used as black-box
  - smart card can use PIN to limit usage
- But still some reasonable usages remain
List of proposals and attacks

- (2002) First WB AES implementation by Chow et. al. [Chow02]
  - IO bijections, linear mixing bijections, external coding
  - broken by BGE cryptanalysis [Bill04]
    - algebraic attack, recovering symmetric key by modelling round function by system of algebraic equations
  - attempt to randomize whitebox primitives, perturbation & random equations added, S-boxes are enc. keys. 4 AES ciphers, major voting for result
  - broken by Mulder et. al. [Mul10]
    - removes perturbations and random equations, attacking on final round removing perturbations, structural decomposition. $2^{17}$ steps
  - broken by Mulder et. al. [Mul12]
    - linear equivalence algorithm used (backward AES-128 compatibility => linear protection has to be inverted in next round), $2^{32}$ steps
- (2011) Protecting white-box AES with dual ciphers [Kar11]
  - broken by our work [Kli13]
    - protection shown to be ineffective
More resources

• Overviews, links

• Crackme challenges

• Whitebox crypto in DRM
Whitebox transform IS used in the wild

• Proprietary DRM systems
  – details are usually not published
  – AES-based functions, keyed hash functions, RSA, ECC...
  – interconnection with surrounding code
• Chow at al. (2002) proposal made at Cloakware
  – firmware protection solution
• Apple’s FairPlay & Brahms attack
  • http://whiteboxcrypto.com/files/2012_MISC_DRM.pdf
• TrojanSpy:Win32/WhiteBox? 😊
• ...
Available practical implementations

DEMO
Demo – WAES

- WAES tables generator
  - configuration options
  - *.h files with pre-computed tables
- WAES cipher implementation
  - compile-in tables
  - tables as memory blob

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/mman.h>
#include <assert.h>

#define AESINVFIRSTTABLE_AUTH_H
#define AESINVFIRSTTABLE_AUTH_H

BYTE invFirstRoundTable_auth[4][4][256] = {
    {0x23, 0x9e, 0x4c, 0x94, 0x4e, 0x32, 0x95, 0x3d, 0x6a, 0x3f, 0x34, 0x99, 0x48, 0x3a, 0x97, 0x3b, 0x46, 0x39, 0x96, 0x45, 0x3c, 0x98, 0x44, 0x3d, 0x9a, 0x49, 0x3e, 0x9b, 0x42, 0x3f, 0x9c, 0x4a, 0x3a, 0x9d, 0x4b, 0x39, 0x9e, 0x4c, 0x38, 0x9f, 0x4d, 0x37, 0xa0, 0x4e, 0x36, 0xa1, 0x4f, 0x35, 0xa2, 0x50, 0x34, 0xa3, 0x51, 0x33, 0xa4, 0x52, 0x32, 0xa5, 0x53, 0x31, 0xa6, 0x54, 0x30, 0xa7, 0x55, 0x2f, 0x9e, 0x4c, 0x94, 0x4e, 0x32, 0x95, 0x3d, 0x6a, 0x3f, 0x34, 0x99, 0x48, 0x3a, 0x97, 0x3b, 0x46, 0x39, 0x96, 0x45, 0x3c, 0x98, 0x44, 0x3d, 0x9a, 0x49, 0x3e, 0x9b, 0x42, 0x3f, 0x9c, 0x4a, 0x3a, 0x9d, 0x4b, 0x39, 0x9e, 0x4c, 0x38, 0x9f, 0x4d, 0x37, 0xa0, 0x4e, 0x36, 0xa1, 0x4f, 0x35, 0xa2, 0x50, 0x34, 0xa3, 0x51, 0x33, 0xa4, 0x52, 0x32, 0xa5, 0x53, 0x31, 0xa6, 0x54, 0x30, 0xa7, 0x55, 0x2f}
```

### Whitebox cryptography - intro

![Image](www.fi.muni.cz/crocs)
WAES performance

- Intel Core i5 M560@2.67GHz

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
<th>Additional info.</th>
<th>OpenSSL result</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate WB AES</td>
<td>8.48 s avg.</td>
<td>100 samples</td>
<td></td>
</tr>
<tr>
<td>throughput, 1 MB random</td>
<td>867.8 KB/s</td>
<td>1.18 s</td>
<td>57283 KB/s</td>
</tr>
<tr>
<td>throughput, 10 MB random</td>
<td>1022.977 KB/s</td>
<td>10.01 s</td>
<td>54179 KB/s</td>
</tr>
<tr>
<td>throughput, 100 MB random</td>
<td>1028.319 KB/s</td>
<td>99.58 s</td>
<td>74744 KB/s</td>
</tr>
<tr>
<td>throughput, 1024 MB random</td>
<td>1124.792 KB/s</td>
<td>932.24 s</td>
<td>63723 KB/s</td>
</tr>
<tr>
<td>throughput, 1 MB null</td>
<td>975 KB/s</td>
<td>1.05 s</td>
<td>93091 KB/s</td>
</tr>
<tr>
<td>throughput, 10 MB null</td>
<td>969.970 KB/s</td>
<td>10.56 s</td>
<td>68821 KB/s</td>
</tr>
<tr>
<td>throughput, 100 MB null</td>
<td>1058.507 KB/s</td>
<td>96.74 s</td>
<td>56356 KB/s</td>
</tr>
<tr>
<td>throughput, 1024 MB null</td>
<td>1050.593 KB/s</td>
<td>998.08 s</td>
<td>57283 KB/s</td>
</tr>
</tbody>
</table>

Table 4.2: Results of the benchmark for whitebox AES generator
BGE attack in progress

recoverQ(); q = 0x88; gamma=0x01;
recoverQ(); self-test: r=5; col=3; (y0, y3); P[0].deltaInv=0x03; alfa (3,0)=0x03
recoverQ(); self-test: r=5; col=3; (y0, y3); P[1].deltaInv=0x01; alfa (3,1)=0x01
recoverQ(); self-test: r=5; col=3; (y0, y3); P[2].deltaInv=0x01; alfa (3,2)=0x01
recoverQ(); self-test: r=5; col=3; (y0, y3); P[3].deltaInv=0x02; alfa (3,3)=0x02
recoverQ(); q = 0x3c; gamma=0x01;

Going to reconstruct encryption key from extracted round keys...
* Round keys extracted from the process, r=3
  0x3d 0x47 0x1e 0x6d 0x80 0x16 0x23 0x7a 0x47 0xfe 0x7e 0x88 0x7d 0x3e 0x44 0x3b
* Round keys extracted from the process, r=4
  0xef 0x68 0x6b 0xda 0x44 0x52 0x71 0x0b 0x5a 0x5b 0x25 0xad 0x41 0x7f 0x3b 0x00
* Round keys extracted from the process, r=5
  0xd4 0x7c 0xca 0x11 0xd1 0x83 0xf2 0xf9 0xc6 0x9d 0xb8 0x15 0xf6 0x87 0xbc 0xbc

Recovering cipher key from round keys...
We have correct Rcon! RconIdx=3
RC=2; previousKey:
  0xf2 0x7a 0x59 0x73
  0xc0 0x96 0x25 0x59
  0x95 0xb9 0x80 0x7f
  0xf2 0x43 0x7a 0x7f
RC=1; previousKey:
  0xa0 0x88 0x29 0x2a
  0xf0 0x54 0xa8 0x6c
  0xf0 0x2c 0x99 0x76
  0x17 0x11 0x89 0x65
RC=0; previousKey:
  0x2b 0x28 0xa6 0x09
  0x76 0x9a 0xf7 0xcf
  0x15 0xd2 0x15 0x4f
  0x16 0xa6 0x88 0x3c

Final result:
  0x2b 0x7c 0x15 0x16 0x26 0xa4 0xd2 0xa6 0x28 0x77 0xf7 0x15 0x88 0xc9 0xcf 0x4f 0x3c

Benchmark finished! Total time = 3s; on average = 1s; clocktime=57.66s;
What’s in our pipeline?

FUTURE WORK
Webpage with implemented proposals

• Obvious next step 😊
• Relevant academic papers didn’t come with implementation 😞
  – true both for proposals and attacks
• Our work provided 2 implementations & 2 attacks
  – we will do remaining soon
• Relevant links
• CrackMe challenges
• http://www.fi.muni.cz/~xsvenda/whiteboxcrypto/
Modifications to W-AES

• Break backward AES compatibility → new cipher
  – but same scheme, strong primitives, key dependency

1. Hash-chain generated round keys
  – noninvertible

2. Key-dependent confusion / S-boxes
  – high variability (13 bytes dependence)

3. Key-dependent diffusion
  – 32x32 -> 128x128 matrix

4. Incorporating of algebraic incompatible operations
  – like in IDEA cipher
Automatic white-box code transformation

- Parse existing source code
- Identify “transformable” operations
  - suitable size of operands
  - no side effects
  - ...
- Transform operations into white-box representation
  - or move to smart card
- Update existing code accordingly
for (i = start; i < end; i += 2) {
    int16_t cbits = 0;
    uint16_t xbits = 0;
    unsigned int xlen = h->xlen;
    unsigned int ext = 0;
    unsigned int x1 = gi->l3_enc[i];
    unsigned int x2 = gi->l3_enc[i + 1];

    assert(gi->l3_enc[i] >= 0);
    assert(gi->l3_enc[i + 1] >= 0);

    if (x1 != 0u) {
        if (gi->xr[i] < 0.0f)
            ext++;
        cbits--;
    }

    if (tableindex > 15u) {
        /* use ESC-words */
        if (x1 >= 15u) {
            uint16_t const linbits_x1 = x1 - 15u;
            assert(linbits_x1 <= h->linmax);
            ext |= linbits_x1 << 1u;
            xbits = linbits;
            x1 = 15u;
        } 

        if (x2 >= 15u) {
            uint16_t const linbits_x2 = x2 - 15u;
            assert(linbits_x2 <= h->linmax);
            ext <<= linbits;
            ext |= linbits_x2;
            xbits += linbits;
            x2 = 15u;
        }
    }

    xlen = 16;
}
Summary

• Computation with encrypted data & function
  – strong whitebox attacker model
• Whitebox cryptography tries to be better than classical obfuscation alone
  – mathematical-level proofs for cryptographic primitives
• Implementation of selected schemes (almost ☺) released
  – published attacks as well

Questions
Thank you for your attention!

Questions